آزمایش ششم

خلاصه ی آزمایش:

استفاده از سخت افزار در رسپبری پای

اهداف آزمایش:

- آشنایی با File I/O
- Time در رسپبری پای
- آشنایی با Wave VCD viewer
- استفاده از GPIO ها در Raspberry pi
- آشنایی با SPI
- آشنایی با i2c

تجهیزات مورد نیاز:

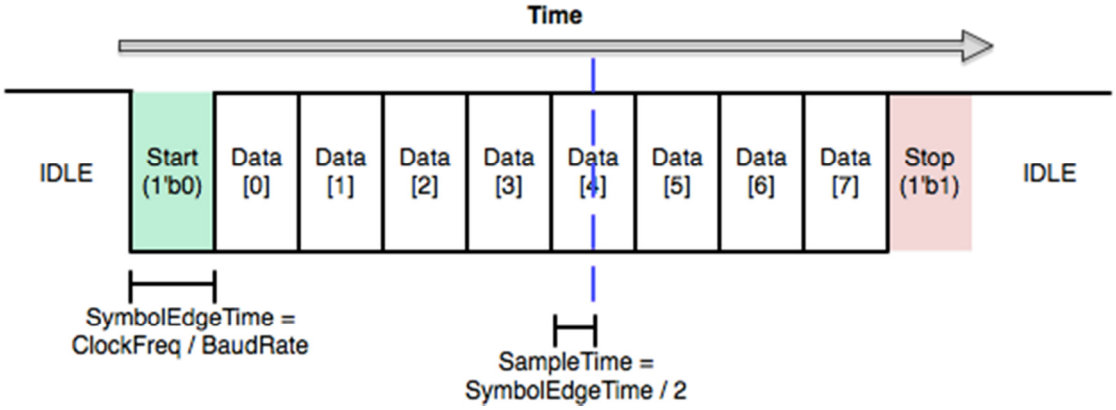
- برد Raspberry pi II و کابل برق آن
- کارت Micro SD
- کامپیوتر (به همراه موس و کیبورد)
- کابل شبکه جهت ارتباط برد با کامپیوتر
- نرم افزار Putty جهت راه اندازی SSH

شرح آزمایش:

1- برنامه ای به زبان C بنویسید که نام یک فایل را به عنوان ورودی بگیرد،آن را کپی کند و با نام copy_ آن را ذخیره کند.

2- برنامه ای بنویسید که یک فایل VCD ایجاد کند که یک موج مربعی با فرکانس ثابت را نمایش دهد.

3- برنامه ای بنویسید که با baud rate 1200 یک داده (string) را به عنوان آرگومان دریافت کند و داده ی string را به صورت سریال روی یک پایه ی ورودی خروجی قرار دهد.



4- برنامه ای بنویسید که نور اتاق را اندازه گیری کند.

5- برنامه ای بنویسید به زبان c که یک بایت را روی spi قرار دهد و بایت خوانده شده را در خروجی نمایش دهد. پایه ی MOSI را به پایه ی MISO متصل کنید و برنامه را اجرا کنید.

وظایف:

1- بخش مطالعه ی این دستور کار به طور کامل مطالعه شود.

مطالعه:

آشنایی با File I/O

Time در رسپبری پای

آشنایی با Wave VCD viewer

استفاده از GPIO ها در Raspberry pi

آشنایی با SPI

آشنایی با i2c

آشنایی با File I/O

Files in C

- #include <stdio.h>
- **FILE** object contains file stream information
- Special files defined in stdio:
  - **stdin**: Standard input
  - **stdout**: Standard ouput
  - **stderr**: Standard error
- **EOF**: end-of-file, a special negative integer constant

Opening a file

```
FILE* fopen(char* filename, char* mode)
```

mode strings	
"r"	Open a file for **reading**. The file must exist.
"w"	Create an empty file for **writing**. If a file with the same name already exists its content is erased and the file is treated as a new empty file.
"a"	**Append** to a file. Writing operations append data at the end of the file. The file is created if it does not exist.

OUPUT

- If successful, returns a pointer to a FILE object
- If fails, returns **NULL**

Closing a file

```
int fclose ( FILE * stream )
```

OUTPUT

- On success, returns 0
- On failure, returns **EOF**

Reading a character from a file

```
int fgetc ( FILE * stream )
```

OUTPUT

- On success, returns the next character
- On failure, returns **EOF** and sets end-of-file indicator

Note: **EOF** < 0; so you can test for failure by checking if the output of **fgetc** is negative

Reading a character from a file

```
UW\n
CSE\n
```

```
FILE *fp = ...
...

while ( (c = fgetc(fp)) != EOF){
    printf("char:'%c'\n",c);
}
```

```
char:'U'
char:'W'
char:'
'
char:'C'
char:'S'
char:'E'
char:'
'
```

Reading a string from a file

```
char * fgets ( char * str, int num, FILE * stream )
```

BEHAVIOR

- Reads at most (*num*-1) characters from the *stream* into *str*
- Null-terminates the string read (adds a '\0' to the end)
- Stops after a newline character is read
- Stops if the end of the file is encountered
  - Caveat: if no characters are read, *str* is not modified

OUTPUT

- On success, a pointer to *str*
- On failure, returns NULL

Are we at the end of a file?

```
int feof ( FILE * stream )
```

OUTPUT

- If at the end of the file, returns a non-zero value
- If not at the end of the file, returns 0

Note: checks the end-of-file indicator which is set by fgets, fgetc, etc.

Reading formatted data from a file

```
int fscanf ( FILE * stream, const char * format, ... )
```

INPUT

- Format string is analogous to **printf** format string
  - %d for integer
  - %c for char
  - %s for string
- Must have an argument for each format specifier

OUTPUT

- On success, returns the number of items read; can be 0 if the pattern doesn't match
- On failure, returns **EOF**

Buffer overruns

- Data is written to locations past the end of the buffer
- Hackers can exploit to execute arbitrary code
- User can *always* create an input longer than **fixed** size of buffer

  Don't use: scanf, fscanf, gets
- Use functions that limit the number of data read

  Use: fgets

Writing a character to a file

```
int fputc ( int character, FILE * stream )
```

OUTPUT / EFFECT

- On success, writes the character to the file and returns the character written
- On failure, returns **EOF** and sets the error indicator

Note: **EOF** < 0; so you can test for failure by checking if the output of **fputc** is negative

Writing a string to a file

```
int fputs ( const char * str, FILE * stream )
```

OUTPUT / EFFECT

- On success, writes the string to the file and returns a non-negative value
- On failure, returns **EOF** and sets the error indicator

Note: **EOF** < 0; so you can test for failure by checking if the output of **fputs** is negative

Writing a formatted string to a file

```
int fprintf ( FILE * stream, const char * format, ... )
```

INPUT
- The format string is same as for **printf**
- Must have an argument for each specifier in the format

OUTPUT / EFFECT
- On success, returns the number of character written
- On failure, returns a negative number

Was there an error?

```
int ferror ( FILE * stream )
```

OUTPUT
- If the error indicator is set, returns a non-zero integer
- Otherwise returns 0

Printing an error description

```
void perror ( const char * str )
```

EFFECT

- Prints a description of the file error prefixed by the supplied string *str* and a ":"
- Can pass **NULL** to just print the error description

Clearing error indicator

```
void clearerr ( FILE * stream );
```

EFFECT

- Clears error indicator
- Clears end-of-file indicator

Going to the beginning of a file

```
void rewind ( FILE * stream );
```

EFFECT
- Moves file pointer to beginning of file
- Resets end-of-file indicator
- Reset error indicator
- Forgets any virtual characters from **ungetc**

Moving to a location

```
int fseek ( FILE * stream, long int offset, int origin )
```

INPUT
- Offset is in bytes
- Origin can be
  - SEEK_SET: beginning of the file
  - SEEK_CUR: current file position
  - SEEK_END: end of the file

OUTPUT / EFFECT
- On success
  - returns 0
  - resets end-of-file indicator
  - forgets any virtual characters from **ungetc**
- On failure, returns 0

Removing a file

```
int remove ( const char * filename )
```

OUTPUT
- On success, returns 0
- On failure, returns a non-zero value

Opening binary files

- Add “b” to the **fopen** mode string
  - “rb” : read a binary file
  - “wb” : write a binary file
  - “ab” : append to a binary file

Writing to binary files

```
size_t fwrite (const void * ptr, size_t size, size_t count, FILE * stream)
```

INPUT
- A *ptr* to an array of elements (or just one)
- The size of each element
- The number of elements

OUTPUT
- Returns the number of elements written
- If return value is different than *count*, there was an error

Reading binary files

```
size_t fread ( void * ptr, size_t size, size_t count, FILE * stream )
```

INPUT
- A *ptr* to some memory of size at least (*size* * *count*)
- The size of each element to read
- The number of elements to read

OUTPUT
- Returns the number of elements read
- If return value is different than *count*, there was an error or the end of the file was reached

Time در رسپبری پای

```
  GNU nano 2.2.6                File: time03.c

#include <stdio.h>
#include <time.h>


int main ()
{
        time_t now = time(NULL);
        printf("time is %s\n", asctime(localtime(&now)));
return 0;
}
```

8.7 *gettimeofday*: Wall-Clock Time

The `gettimeofday` system call gets the system's wall-clock time. It takes a pointer to a `struct timeval` variable. This structure represents a time, in seconds, split into two fields. The `tv_sec` field contains the integral number of seconds, and the `tv_usec` field contains an additional number of microseconds. This `struct timeval` value represents the number of seconds elapsed since the start of the *UNIX epoch*, on midnight UTC on January 1, 1970. The `gettimeofday` call also takes a second argument, which should be `NULL`. Include `<sys/time.h>` if you use this system call.

The number of seconds in the UNIX epoch isn't usually a very handy way of representing dates. The `localtime` and `strftime` library functions help manipulate the return value of `gettimeofday`. The `localtime` function takes a pointer to the number of seconds (the `tv_sec` field of `struct timeval`) and returns a pointer to a `struct tm` object. This structure contains more useful fields, which are filled according to the local time zone:

- `tm_hour`, `tm_min`, `tm_sec`—The time of day, in hours, minutes, and seconds.
- `tm_year`, `tm_mon`, `tm_day`—The year, month, and date.
- `tm_wday`—The day of the week. Zero represents Sunday.
- `tm_yday`—The day of the year.
- `tm_isdst`—A flag indicating whether daylight savings time is in effect.

The `strftime` function additionally can produce from the `struct tm` pointer a customized, formatted string displaying the date and time. The format is specified in a manner similar to `printf`, as a string with embedded codes indicating which time fields to include. For example, this format string

```
"%Y-%m-%d %H:%M:%S"
```

specifies the date and time in this form:

```
2001-01-14 13:09:42
```

Pass `strftime` a character buffer to receive the string, the length of that buffer, the format string, and a pointer to a `struct tm` variable. See the `strftime` man page for a complete list of codes that can be used in the format string. Notice that neither `localtime` nor `strftime` handles the fractional part of the current time more precise than 1 second (the `tv_usec` field of `struct timeval`). If you want this in your formatted time strings, you'll have to include it yourself.

Include `<time.h>` if you call `localtime` or `strftime`.

The function in Listing 8.6 prints the current date and time of day, down to the millisecond.

Listing 8.6 **(*print-time.c*) Print Date and Time**

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>

void print_time ()
{
  struct timeval tv;
  struct tm* ptm;
  char time_string[40];
  long milliseconds;

  /* Obtain the time of day, and convert it to a tm struct.  */
  gettimeofday (&tv, NULL);
  ptm = localtime (&tv.tv_sec);
  /* Format the date and time, down to a single second.  */
  strftime (time_string, sizeof (time_string), "%Y-%m-%d %H:%M:%S", ptm);
  /* Compute milliseconds from microseconds.  */
  milliseconds = tv.tv_usec / 1000;
  /* Print the formatted time, in seconds, followed by a decimal point
     and the milliseconds.  */
  printf ("%s.%03ld\n", time_string, milliseconds);
}
```

8.10 *nanosleep*: High-Precision Sleeping

The `nanosleep` system call is a high-precision version of the standard UNIX `sleep` call. Instead of sleeping an integral number of seconds, `nanosleep` takes as its argument a pointer to a `struct timespec` object, which can express time to nanosecond precision. However, because of the details of how the Linux kernel works, the actual precision provided by `nanosleep` is 10 milliseconds—still better than that afforded by `sleep`. This additional precision can be useful, for instance, to schedule frequent operations with short time intervals between them.

The `struct timespec` structure has two fields: `tv_sec`, the integral number of seconds, and `tv_nsec`, an additional number of milliseconds. The value of `tv_nsec` must be less than 10^9.

The `nanosleep` call provides another advantage over `sleep`. As with `sleep`, the delivery of a signal interrupts the execution of `nanosleep`, which sets `errno` to `EINTR` and returns –1. However, `nanosleep` takes a second argument, another pointer to a `struct timespec` object, which, if not null, is filled with the amount of time remaining (that is, the difference between the requested sleep time and the actual sleep time). This makes it easy to resume the sleep operation.

The function in Listing 8.8 provides an alternate implementation of `sleep`. Unlike the ordinary system call, this function takes a floating-point value for the number of seconds to sleep and restarts the sleep operation if it's interrupted by a signal.

Listing 8.8 **(*better_sleep.c*) High-Precision Sleep Function**

```
#include <errno.h>
#include <time.h>

int better_sleep (double sleep_time)
{
  struct timespec tv;
  /* Construct the timespec from the number of whole seconds...  */
  tv.tv_sec = (time_t) sleep_time;
  /* ... and the remainder in nanoseconds.  */
  tv.tv_nsec = (long) ((sleep_time - tv.tv_sec) * 1e+9);

  while (1)
  {
    /* Sleep for the time specified in tv.  If interrupted by a
       signal, place the remaining time left to sleep back into tv.  */
    int rval = nanosleep (&tv, &tv);
    if (rval == 0)
      /* Completed the entire sleep time; all done.  */
      return 0;
    else if (errno == EINTR)
      /* Interrupted by a signal.  Try again.  */
      continue;
    else
      /* Some other error; bail out.  */
      return rval;
  }
  return 0;
}
```

8.13 *setitimer*: Setting Interval Timers

The `setitimer` system call is a generalization of the `alarm` call. It schedules the delivery of a signal at some point in the future after a fixed amount of time has elapsed.

A program can set three different types of timers with `setitimer`:

- If the timer code is `ITIMER_REAL`, the process is sent a `SIGALRM` signal after the specified wall-clock time has elapsed.
- If the timer code is `ITIMER_VIRTUAL`, the process is sent a `SIGVTALRM` signal after the process has executed for the specified time. Time in which the process is not executing (that is, when the kernel or another process is running) is not counted.
- If the timer code is `ITIMER_PROF`, the process is sent a `SIGPROF` signal when the specified time has elapsed either during the process's own execution or the execution of a system call on behalf of the process.

The first argument to `setitimer` is the timer code, specifying which timer to set. The second argument is a pointer to a `struct itimerval` object specifying the new settings for that timer. The third argument, if not null, is a pointer to another `struct itimerval` object that receives the old timer settings.

A `struct itimerval` variable has two fields:

- `it_value` is a `struct timeval` field that contains the time until the timer next expires and a signal is sent. If this is 0, the timer is disabled.
- `it_interval` is another `struct timeval` field containing the value to which the timer will be reset after it expires. If this is 0, the timer will be disabled after it expires. If this is nonzero, the timer is set to expire repeatedly after this interval.

The `struct timeval` type is described in Section 8.7, "`gettimeofday`: Wall-Clock Time."

The program in Listing 8.11 illustrates the use of `setitimer` to track the execution time of a program. A timer is configured to expire every 250 milliseconds and send a `SIGVTALRM` signal.

Listing 8.11 **(*itemer.c*) Timer Example**

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>

void timer_handler (int signum)
{
  static int count = 0;
  printf ("timer expired %d times\n", ++count);
}

int main ()
{
  struct sigaction sa;
  struct itimerval timer;

  /* Install timer_handler as the signal handler for SIGVTALRM.  */
  memset (&sa, 0, sizeof (sa));
  sa.sa_handler = &timer_handler;
  sigaction (SIGVTALRM, &sa, NULL);

  /* Configure the timer to expire after 250 msec...  */
  timer.it_value.tv_sec = 0;
  timer.it_value.tv_usec = 250000;
  /* ... and every 250 msec after that.  */
  timer.it_interval.tv_sec = 0;
  timer.it_interval.tv_usec = 250000;
  /* Start a virtual timer.  It counts down whenever this process is
     executing.  */
  setitimer (ITIMER_VIRTUAL, &timer, NULL);

  /* Do busy work.  */
  while (1);
}
```

Listing B.2 **(*timestamp.c*) Append a Timestamp to a File**

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

/* Return a character string representing the current date and time.  */

char* get_timestamp ()
{
  time_t now = time (NULL);
  return asctime (localtime (&now));
}

int main (int argc, char* argv[])
{
  /* The file to which to append the timestamp.  */
  char* filename = argv[1];
  /* Get the current timestamp.  */
  char* timestamp = get_timestamp ();
  /* Open the file for writing.  If it exists, append to it;
     otherwise, create a new file.  */
  int fd = open (filename, O_WRONLY | O_CREAT | O_APPEND, 0666);
  /* Compute the length of the timestamp string.  */
  size_t length = strlen (timestamp);
  /* Write the timestamp to the file.  */
  write (fd, timestamp, length);
  /* All done.  */
  close (fd);
  return 0;
}
```

تایمینگ دقیق در رسپبری:

Anatomy of the Clock

The first mistake most people make when doing timing is to use functions like `gettimeofday()`, `GetSystemTime()`, etc. These functions return what is called "wall time"... time that corresponds to a calendar date/time. These clocks suffer from the follow limitations:

1. They have a low resolution: "High-performance" timing, by my definition, requires clock resolutions into the microseconds or better.
2. They can jump forwards and backwards in time: Computer clocks all tick at slightly different rates, which causes the time to drift. Most systems have NTP enabled which periodically adjusts the system clock to keep them in sync with "actual" time. The adjustment can cause the clock to suddenly jump forward (artificially inflating your timing numbers) or jump backwards (causing your timing calculations to go negative or hugely positive).

Using the Clock

To use the clock, you need access to 2 pieces of information: the clock value, and the rate (frequency) at which it increments. Once you have that information, it's trivial to calculate the duration between 2 successive reads of the clock.

For example, consider the following:

- `start` equals the clock value at the beginning of the interval to measure.
- `end` equals the clock value at the end of the interval.
- `frequency` equals the frequency that the clock increments per second.

Calculating the duration of the interval (in seconds) is as simple as:

```
duration = (end - start) / frequency
```

`duration` will equal the floating-point interval time in seconds (e.g. 0.000237 = 237us).

A common use case I've seen is wanting to measure the duration between successive calls to a function. Here's one way to implement that:

```
float freq = (float) get_frequency();

Foo()
{
    unsigned now = read_clock();
    float duration = (float)(now - last) / (float)freq;
    last = now;

    /* Your code here */
}
```

Linux Clock

POSIX.1b defines realtime clock methods that you'll find on most *NIX systems (the full spec can be viewed HERE). Specifically, you want to use `clock_getres()` and `clock_gettime()`. `clock_getres()` returns the resolution (frequency) of the clock, and `clock_gettime()` returns the current value of the clock. Most systems implement the `CLOCK_MONOTONIC` type, which provides a frequency-stable, monotonically-increasing counter. The resolution of `CLOCK_MONOTONIC` is high on the 2.6 kernel, in my experience. I recommend using this clock when building a high-performance timing solutions on Linux.

The methods are defined in `time.h`, and you need to link against librt (pass '-lrt' to gcc). The prototypes for the functions are:

```
int clock_getres(clockid_t clock_id, struct timespec *res);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
```

```
  GNU nano 2.2.6                                File: time04.c

#include <stdio.h>
#include <time.h>


int main ( int argc , char *argv[])
{
        struct timespec tv,tv_start,tv_end;
        double sleep_time;
        double waited_time;
        if (argc < 2) {
                printf("usage: ...\n");
                return -1;
        }
        sleep_time = 1.0/atoi(argv[1]);
        printf("freq is %0.4f\n",1/sleep_time);
        printf("delay time is %0.3f ms\n", sleep_time*1000.0);
        clock_gettime(CLOCK_REALTIME,&tv_start);
        waited_time = 0;
        while( waited_time < sleep_time*1000000000){
                clock_gettime(CLOCK_REALTIME,&tv_end);
                waited_time = ((tv_end.tv_sec - tv_start.tv_sec)*1000000000);
                waited_time += (tv_end.tv_nsec - tv_start.tv_nsec);
        }
        printf("delay is %0.4f ms\n",(waited_time)/1000000);
        return 0;
}
```

```
root@raspberrypi:/home/pi/session06/time# gcc -o time04 time04.c -lrt
root@raspberrypi:/home/pi/session06/time# ./time04 14400
freq is 14400.0000
delay time is 0.069 ms
delay is 0.0698 ms
root@raspberrypi:/home/pi/session06/time# ./time04 19200
freq is 19200.0000
delay time is 0.052 ms
delay is 0.0522 ms
root@raspberrypi:/home/pi/session06/time# ./time04 38400
freq is 38400.0000
delay time is 0.026 ms
delay is 0.0262 ms
root@raspberrypi:/home/pi/session06/time#
```

clock_gettime(3) - Linux man page

Name

clock_getres, clock_gettime, clock_settime - clock and time functions

Synopsis

#include <time.h>

int clock_getres(clockid_t *clk_id*, **struct timespec** **res*);
int clock_gettime(clockid_t *clk_id*, **struct timespec** **tp*);
int clock_settime(clockid_t *clk_id*, **const struct timespec** **tp*);

Description

The function **clock_getres**() finds the resolution (precision) of the specified clock *clk_id*, and, if *res* is non-NULL, stores it in the struct timespec pointed to by *res*. The resolution of clocks depends on the implementation and cannot be configured by a particular process. If the time value pointed to by the argument *tp* of **clock_settime**() is not a multiple of *res*, then it is truncated to a multiple of *res*.

The functions **clock_gettime**() and **clock_settime**() retrieve and set the time of the specified clock *clk_id*.

The *res* and *tp* arguments are **timespec** structs, as specified in <*time.h*>:

```
struct timespec {
        time_t   tv_sec;        /* seconds */
        long     tv_nsec;       /* nanoseconds */
};
```

The *clk_id* argument is the identifier of the particular clock on which to act. A clock may be system-wide and hence visible for all processes, or per-process if it measures time only within a single process.

All implementations support the system-wide realtime clock, which is identified by **CLOCK_REALTIME**. Its time represents seconds and nanoseconds since the Epoch. When its time is changed, timers for a relative interval are unaffected, but timers for an absolute point in time are affected.

More clocks may be implemented. The interpretation of the corresponding time values and the effect on timers is unspecified.

Sufficiently recent versions of GNU libc and the Linux kernel support the following clocks:

CLOCK_REALTIME
System-wide realtime clock. Setting this clock requires appropriate privileges.
CLOCK_MONOTONIC
Clock that cannot be set and represents monotonic time since some unspecified starting point.
CLOCK_PROCESS_CPUTIME_ID
High-resolution per-process timer from the CPU.
CLOCK_THREAD_CPUTIME_ID
Thread-specific CPU-time clock.

Return Value

clock_gettime(), **clock_settime**() and **clock_getres**()
return 0 for success, or -1 for failure (in which case
errno is set appropriately).

Errors

EFAULT

tp points outside the accessible address space.

EINVAL

The *clk_id* specified is not supported on this system.

EPERM

clock_settime() does not have permission to set the
clock indicated.

Note

Most systems require the program be linked with the librt library to use these functions.

دو برنامه ی زیر را بررسی کنید و به تفاوت های آن دقت کنید.

```
 GNU nano 2.2.6                                File: time01.c

#include <time.h>
#include <stdio.h>

int main()
{
    clock_t t0 = clock() ;
    clock_t t1 = t0 ;
    while( t0 == t1 )
    {
        t1 = clock() ;
    }
    printf( "Clock granularity = %lf seconds\n", (double)(t1 - t0)/(double)(CLOCKS_PER_SEC) ) ;
    return 0 ;
}
```

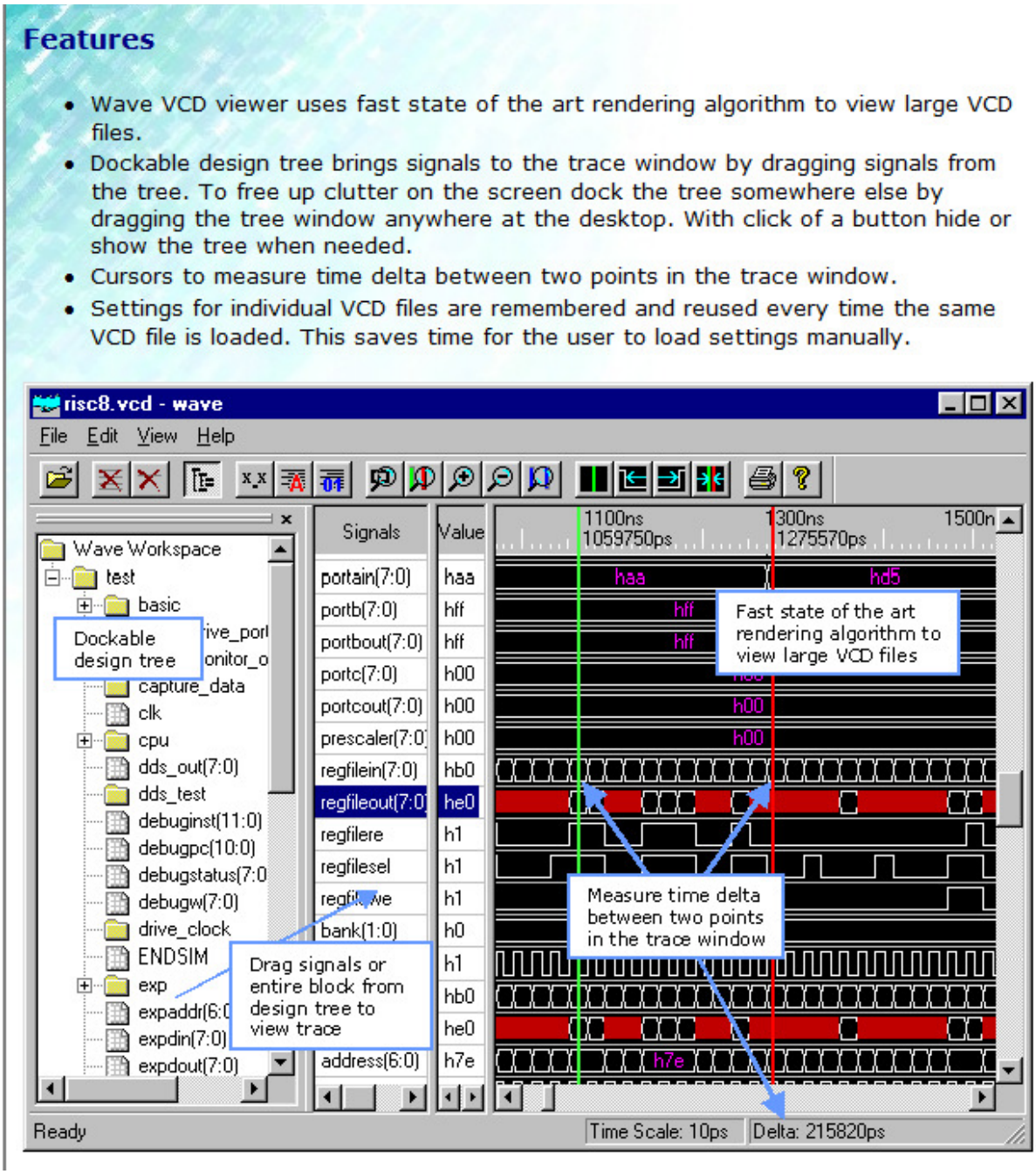
```
root@raspberrypi:/home/pi/session06/time# gcc -o time01 time01.c
root@raspberrypi:/home/pi/session06/time# ./time01
Clock granularity = 0.010000 seconds
root@raspberrypi:/home/pi/session06/time#
```

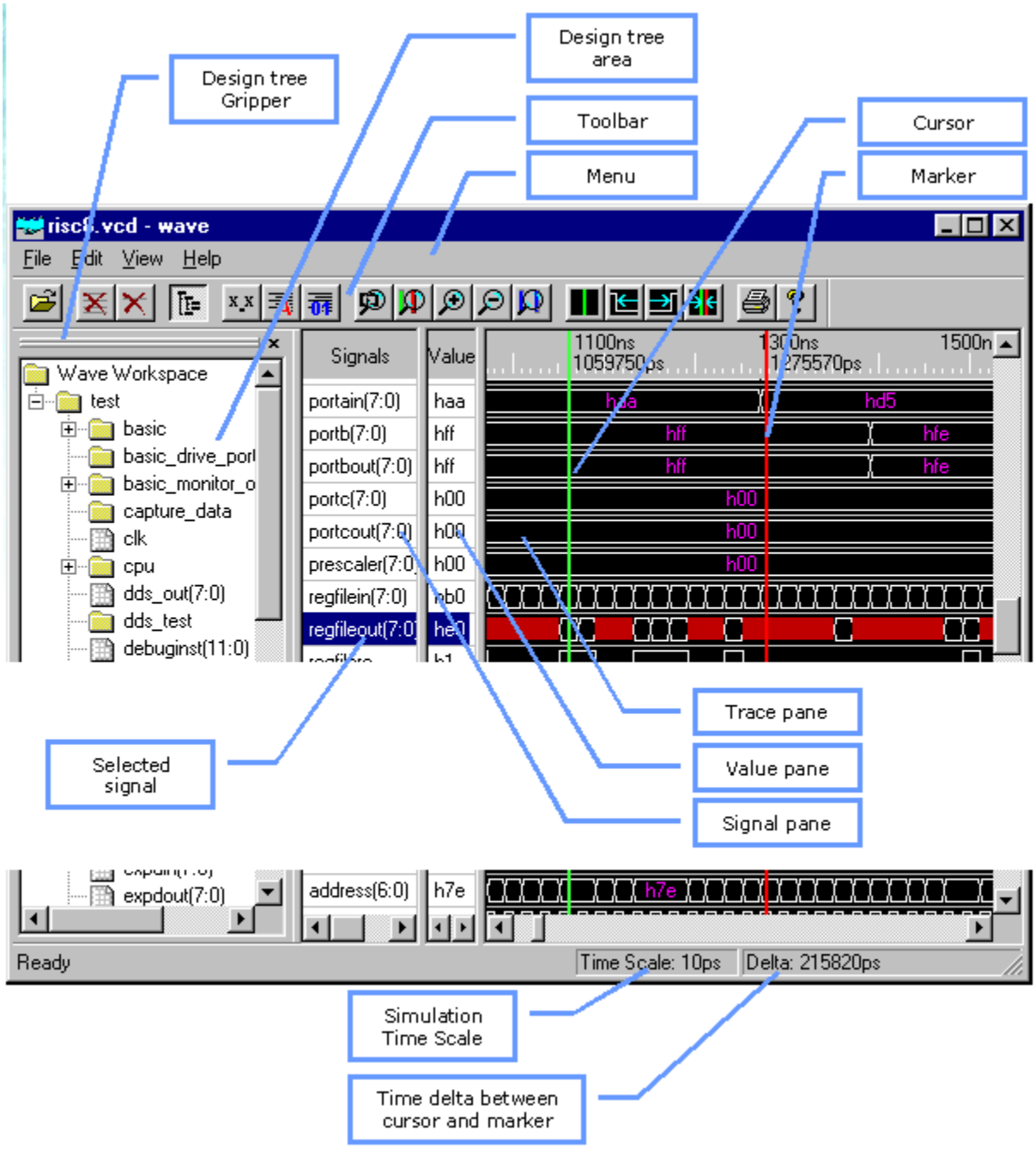
```
 GNU nano 2.2.6                                File: time02.c

#include <time.h>
#include <stdio.h>

int main()
{
    struct timespec t0 ;
    struct timespec t1 ;
    clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &t0) ;
    t1 = t0 ;
    while( t0.tv_nsec == t1.tv_nsec )
    {
        clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &t1) ;
    }
    printf( "Clock granularity = %lf seconds\n", (double)(t1.tv_nsec - t0.tv_nsec)/1000000000 ) ;

    struct timespec res ;
    clock_getres( CLOCK_PROCESS_CPUTIME_ID, &res ) ;
    printf( "CPUTIME resolution = %.15lf seconds\n", (double)(res.tv_nsec)/1000000000) ;

    return 0 ;
}
```

```
root@raspberrypi:/home/pi/session06/time# gcc -o time02 time02.c -lrt
root@raspberrypi:/home/pi/session06/time# ./time02
Clock granularity = 0.000009 seconds
CPUTIME resolution = 0.000000001000000 seconds
root@raspberrypi:/home/pi/session06/time#
```

آشنایی با Wave VCD viewer

ابتدا نرم افزار را **در کامپیوتر** نصب کنید.

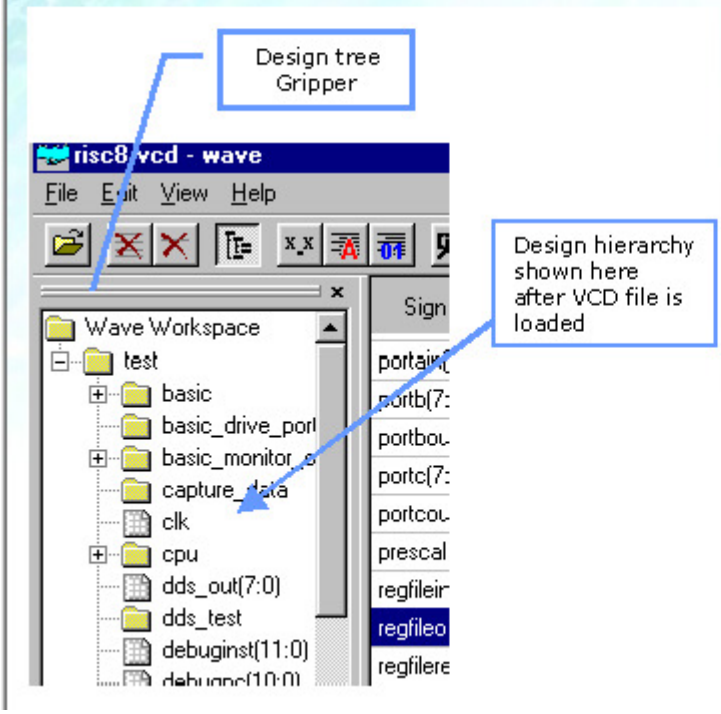



Design tree Gripper
Design tree area
Toolbar
Menu
Cursor
Marker
risc5.vcd - wave
File Edit View Help
Wave Workspace
test
basic
basic_drive_port
basic_monitor_o
capture_data
clk
cpu
dds_out(7:0)
dds_test
debuginst(11:0)
Signals
Value
portain(7:0)
portb(7:0)
portbout(7:0)
portc(7:0)
portcout(7:0)
prescaler(7:0)
regfilein(7:0)
regfileout(7:0)
1100ns
1059750ps
1300ns
1275570ps
1500n
Selected signal
Trace pane
Value pane
Signal pane
expdout(7:0)
address(6:0)
h7e
Ready
Time Scale: 10ps
Delta: 215820ps
Simulation Time Scale
Time delta between cursor and marker

Loading a VCD File

Select menu item **File -> Open VCD** and a file selection dialog box will open. Select the VCD file to load in the dialog box to open and the VCD file will be analyzed. This may take a few minutes to complete depending on the size of the VCD file.

As an alternative, you can press the **Ctrl-O** key from the keyboard or click on the button on the toolbar.



آشنایی با فایل VCD :

Structure/Syntax [edit]

This section requires expansion.
(May 2008)

The VCD file comprises a header section with date, simulator, and timescale information; a variable definition section; and a value change section, in that order. The sections are not explicitly delineated within the file, but are identified by the inclusion of keywords belonging to each respective section.

VCD keywords are marked by a leading **$** (but variable identifiers can also start with a **$**). In general every keyword starts a section which is terminated by an **$end** keyword.

All VCD tokens are delineated by whitespace. Data in the VCD file is case sensitive.

Header section [edit]

The header section of the VCD file includes a timestamp, a simulator version number, and a timescale, which maps the time increments listed in the value change section to simulation time units.

Variable definition section [edit]

The variable definition section of the VCD file contains scope information as well as lists of signals instantiated in a given scope.

Each variable is assigned an arbitrary, compact ASCII identifier for use in the value change section. The identifier is composed of printable ASCII characters from **!** to ~ (decimal 33 to 126). Several variables can share an identifier if the simulator determines that they will always have the same value.

The scope type definitions closely follow Verilog concepts, and include the types *module*, *task*, *function*, and *fork*.

$dumpvars section [edit]

The section beginning with $dumpvars keyword contains initial values of all variables dumped.

Value change section [edit]

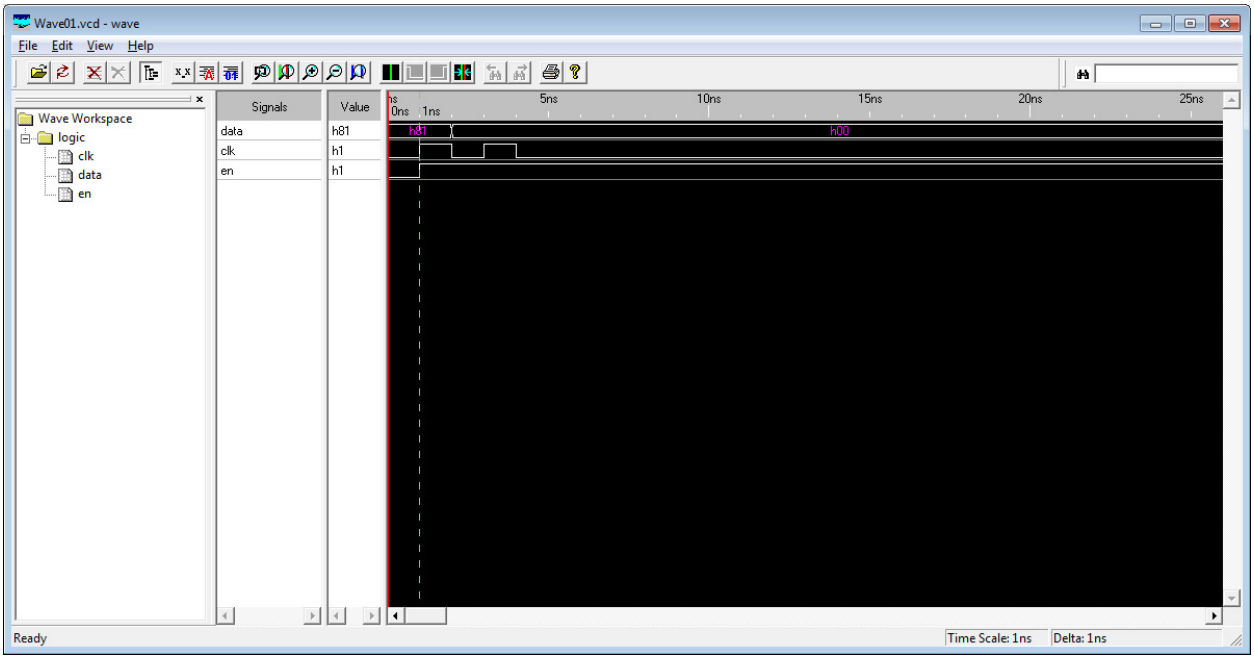
The value change section contains a series of time-ordered value changes for the signals in a given simulation model. For scalar (single bit) signal the format is signal value denoted by 0 or 1 followed immediately by the signal identifier with no space between the value and the signal identifier. For vector (multi-bit) signals the format is signal value denoted by letter 'b' or 'B' followed by the value in binary format followed by space and then the signal identifier. Value for real variables is denoted by letter 'r' or 'R' followed by the data using %.16g printf() format followed by space and then the variable identifier.

```
$date
   Date text. For example: November 16, 2015.
$end
$version
   VCD generator tool version info text.
$end
$comment
   Any comment text.
$end
$timescale 1ns $end
```

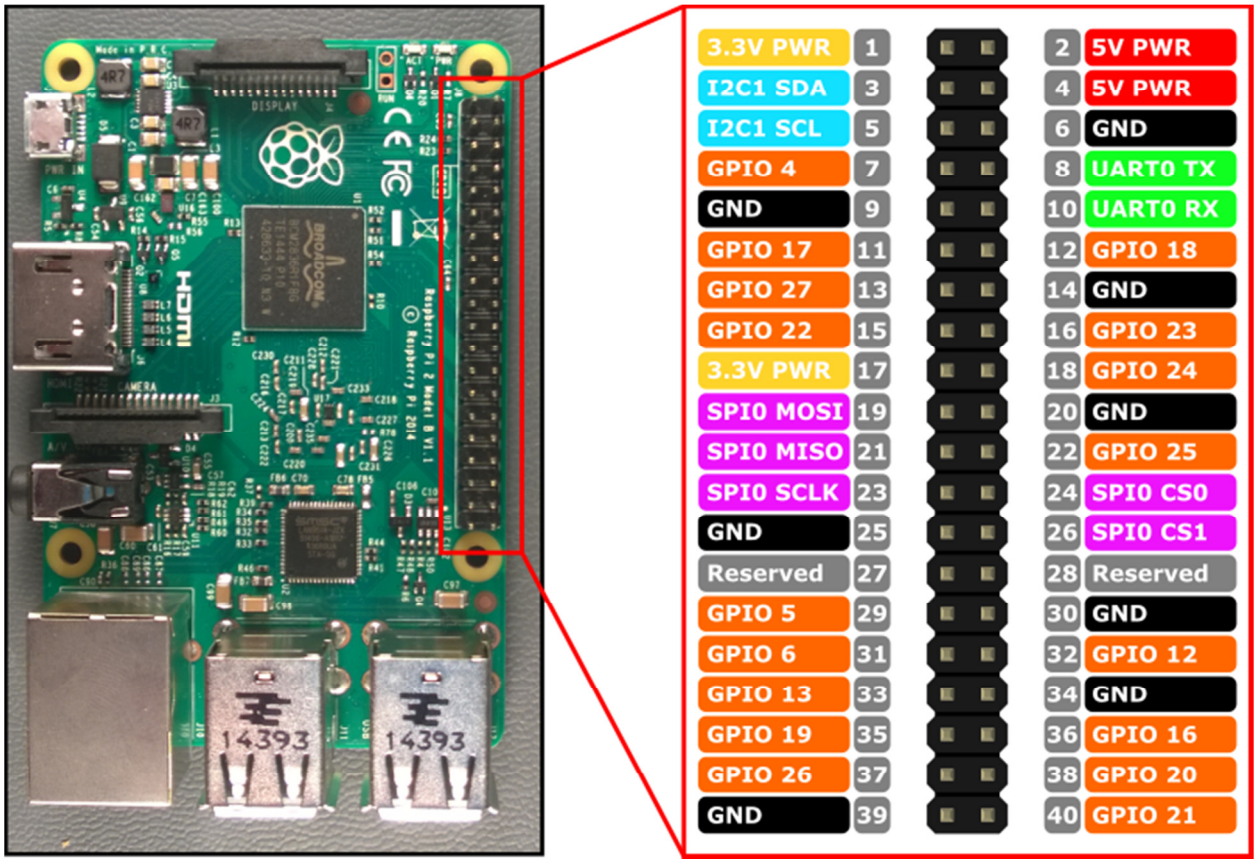
```
$scope module logic $end
$var wire 8 # data $end
$var wire 1 $ clk $end
$var wire 1 % en $end
$upscope $end
$enddefinitions $end
```

```
$dumpvars
bxxxxxxxx #
x$
0%
$end
```

```
#0
b10000001 #
0$
#1
1$
1%
#2
b0 #
0$
#3
1$
#4
0$
#5
```


Wave01.vcd - wave
File Edit View Help
Wave Workspace
logic
clk
data
en
Signals
Value
data
clk
en
h81
h1
h1
0ns
1ns
5ns
10ns
15ns
20ns
25ns
h81
h00
Ready
Time Scale: 1ns
Delta: 1ns

استفاده از GPIO ها در Raspberry pi



دو روش برای کار کردن با GPIO وجود دارد .

Using the GPIO Pins

- There are two different methods to read or write these pins using Linux
  - Creating a file-type access in the file system
  - Write/read memory addresses allocated to the GPIO peripheral of the SoC using pointers
    - Memory locations can be found in the datasheet for the BCM2835
- We can use the Wiring library to help with both

روش اول :

Understanding /sys/class/gpio/

- In Linux everything is a file: /dev/ttyUSB0, /sys/class/net/eth0/address, /dev/mmcblk0p2,…
- sysfs in a kernel module providing a virtual file system for device access at /sys/class
  - provides a way for users (or code in the *user-space*) to interact with devices at the system (kernel) level
- A demo
- Advantages / Disadvantage
  - Allows conventional access to pins from userspace
  - Always involves mode switch to kernel, action in kernel, mode switch to use, and could have a context switch
  - Much slower the digitalWrite()/digitalRead() of Arduino

```
root@raspberrypi:/home/pi# echo 25 > /sys/class/gpio/export
root@raspberrypi:/home/pi# cd /sys/class/gpio/gpio25
root@raspberrypi:/sys/class/gpio/gpio25# ls
active_low  device  direction  edge  subsystem uevent value
root@raspberrypi:/sys/class/gpio/gpio25# echo out > direction
root@raspberrypi:/sys/class/gpio/gpio25# echo 1 > value
root@raspberrypi:/sys/class/gpio/gpio25# echo 0 > value
root@raspberrypi:/sys/class/gpio/gpio25# _
```

این برنامه فقط برای آشنایی بیشتر است و لازم نیست در کلاس این برنامه را اجرا کنید.

```
  GNU nano 2.2.6                                    File: blinky_sysfs.c

#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define IN  0
#define OUT 1

#define LOW  0
#define HIGH 1

#define PIN  24 /* P1-18 */
#define POUT 25  /* P1-22 */
```

```
static int
GPIOExport(int pin)
{
#define BUFFER_MAX 3
        char buffer[BUFFER_MAX];
        ssize_t bytes_written;
        int fd;

        fd = open("/sys/class/gpio/export", O_WRONLY);
        if (-1 == fd) {
                fprintf(stderr, "Failed to open export for writing!\n");
                return(-1);
        }

        bytes_written = snprintf(buffer, BUFFER_MAX, "%d", pin);
        write(fd, buffer, bytes_written);
        close(fd);
        return(0);
}
```

```c
static int
GPIOUnexport(int pin)
{
        char buffer[BUFFER_MAX];
        ssize_t bytes_written;
        int fd;

        fd = open("/sys/class/gpio/unexport", O_WRONLY);
        if (-1 == fd) {
                fprintf(stderr, "Failed to open unexport for writing!\n");
                return(-1);
        }

        bytes_written = snprintf(buffer, BUFFER_MAX, "%d", pin);
        write(fd, buffer, bytes_written);
        close(fd);
        return(0);
}
```

```c
static int
GPIODirection(int pin, int dir)
{
        static const char s_directions_str[]  = "in\0out";

#define DIRECTION_MAX 35
        char path[DIRECTION_MAX];
        int fd;

        snprintf(path, DIRECTION_MAX, "/sys/class/gpio/gpio%d/direction", pin);
        fd = open(path, O_WRONLY);
        if (-1 == fd) {
                fprintf(stderr, "Failed to open gpio direction for writing!\n");
                return(-1);
        }

        if (-1 == write(fd, &s_directions_str[IN == dir ? 0 : 3], IN == dir ? 2 : 3)) {
                fprintf(stderr, "Failed to set direction!\n");
                return(-1);
        }

        close(fd);
        return(0);
}
```

```
static int
GPIORead(int pin)
{
#define VALUE_MAX 30
        char path[VALUE_MAX];
        char value_str[3];
        int fd;

        snprintf(path, VALUE_MAX, "/sys/class/gpio/gpio%d/value", pin);
        fd = open(path, O_RDONLY);
        if (-1 == fd) {
                fprintf(stderr, "Failed to open gpio value for reading!\n");
                return(-1);
        }

        if (-1 == read(fd, value_str, 3)) {
                fprintf(stderr, "Failed to read value!\n");
                return(-1);
        }

        close(fd);

        return(atoi(value_str));
}
```

```
static int
GPIOWrite(int pin, int value)
{
        static const char s_values_str[] = "01";

        char path[VALUE_MAX];
        int fd;

        snprintf(path, VALUE_MAX, "/sys/class/gpio/gpio%d/value", pin);
        fd = open(path, O_WRONLY);
        if (-1 == fd) {
                fprintf(stderr, "Failed to open gpio value for writing!\n");
                return(-1);
        }

        if (1 != write(fd, &s_values_str[LOW == value ? 0 : 1], 1)) {
                fprintf(stderr, "Failed to write value!\n");
                return(-1);
        }

        close(fd);
        return(0);
}
```

```c
int
main(int argc, char *argv[])
{
        int repeat = 10;
        /* Enable GPIO pins */
        if (-1 == GPIOExport(POUT) || -1 == GPIOExport(PIN))
                return(1);

        /* Set GPIO directions */
        if (-1 == GPIODirection(POUT, OUT) || -1 == GPIODirection(PIN, IN))
                return(2);

        do {
                /* Write GPIO value */
                if (-1 == GPIOWrite(POUT, repeat % 2))
                        return(3);

                /* Read GPIO value */
                printf("I'm reading %d in GPIO %d\n", GPIORead(PIN), PIN);

                usleep(500 * 1000);
        }
        while (repeat--);

        /* Disable GPIO pins */
        if (-1 == GPIOUnexport(POUT) || -1 == GPIOUnexport(PIN))
                return(4);
        return(0);
}
```

روش دوم:

این برنامه فقط برای آشنایی بیشتر است و لازم نیست در کلاس این برنامه را اجرا کنید.

```
 GNU nano 2.2.6                                          File: gpio_direct.c

#define BCM2708_PERI_BASE        0x3F000000
#define GPIO_BASE                (BCM2708_PERI_BASE + 0x200000) /* GPIO controller */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

#define PAGE_SIZE (4*1024)
#define BLOCK_SIZE (4*1024)

int  mem_fd;
void *gpio_map;

// I/O access
volatile unsigned *gpio;


// GPIO setup macros. Always use INP_GPIO(x) before using OUT_GPIO(x) or SET_GPIO_ALT(x,y)
#define INP_GPIO(g) *(gpio+((g)/10)) &= ~(7<<(((g)%10)*3))
#define OUT_GPIO(g) *(gpio+((g)/10)) |=  (1<<(((g)%10)*3))
#define SET_GPIO_ALT(g,a) *(gpio+(((g)/10))) |= (((a)<=3?(a)+4:(a)==4?3:2)<<(((g)%10)*3))

#define GPIO_SET *(gpio+7)  // sets   bits which are 1 ignores bits which are 0
#define GPIO_CLR *(gpio+10) // clears bits which are 1 ignores bits which are 0

#define GET_GPIO(g) (*(gpio+13)&(1<<g)) // 0 if LOW, (1<<g) if HIGH

#define GPIO_PULL *(gpio+37) // Pull up/pull down
#define GPIO_PULLCLK0 *(gpio+38) // Pull up/pull down clock
```

```
void setup_io();

void printButton(int g)
{
  if (GET_GPIO(g)) // !=0 <-> bit is 1 <- port is HIGH=3.3V
    printf("Button pressed!\n");
  else // port is LOW=0V
    printf("Button released!\n");
}
```

```
int main(int argc, char **argv)
{
  int g,rep;
  // Set up gpi pointer for direct register access
  setup_io();

 /************************************************************************\
  * You are about to change the GPIO settings of your computer.          *
  * Mess this up and it will stop working!                               *
  * It might be a good idea to 'sync' before running this program        *
  * so at least you still have your code changes written to the SD-card! *
 \************************************************************************/

  // Set GPIO pins 7-11 to output
  for (g=7; g<=11; g++)
  {
    INP_GPIO(g); // must use INP_GPIO before we can use OUT_GPIO
    OUT_GPIO(g);
  }
  for (rep=0; rep<10; rep++)
  {
    for (g=7; g<=11; g++)
     {
       GPIO_SET = 1<<8;
       usleep(10);
     }
     for (g=7; g<=11; g++)
     {
       GPIO_CLR = 1<<8;
       usleep(10);
     }
  }
  return 0;
}_
```

```
void setup_io()
{
   /* open /dev/mem */
   if ((mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
      printf("can't open /dev/mem \n");
      exit(-1);
   }
   /* mmap GPIO */
   gpio_map = mmap(
      NULL,             //Any adddress in our space will do
      BLOCK_SIZE,       //Map length
      PROT_READ|PROT_WRITE,// Enable reading & writting to mapped memory
      MAP_SHARED,       //Shared with other processes
      mem_fd,           //File to map
      GPIO_BASE         //Offset to GPIO peripheral
   );

   close(mem_fd); //No need to keep mem_fd open after mmap

   if (gpio_map == MAP_FAILED) {
      printf("mmap error %d\n", (int)gpio_map);//errno also set!
      exit(-1);
   }

   // Always use volatile pointer!
   gpio = (volatile unsigned *)gpio_map;


}
```

روشی که برای تست ها در این جلسه از آن استفاده می کنیم، استفاده از کتابخانه ی pigpio است .

ابتدا باید این کتابخانه را روی رسپبری نصب کنیم. برای این کار باید ابتدا فولدر pigpio را به رسپبری منتقل کنید .

```
root@raspberrypi:/home/pi/session06# cd pigpio/
root@raspberrypi:/home/pi/session06/pigpio# make
gcc -O3 -Wall -pthread -fpic -c -o pigpio.o pigpio.c
gcc -O3 -Wall -pthread -fpic -c -o command.o command.c
gcc -shared -o libpigpio.so pigpio.o command.o
strip --strip-unneeded libpigpio.so
size     libpigpio.so
   text    data     bss     dec     hex filename
 205062    4772  673352  883186   d79f2 libpigpio.so
gcc -O3 -Wall -pthread -fpic -c -o pigpiod_if.o pigpiod_if.c
gcc -shared -o libpigpiod_if.so pigpiod_if.o command.o
strip --strip-unneeded libpigpiod_if.so
size     libpigpiod_if.so
   text    data     bss     dec     hex filename
```

```
root@raspberrypi:/home/pi/session06/pigpio# make install
install -m 0755 -d                  /opt/pigpio/cgi
install -m 0755 -d                  /usr/local/include
install -m 0644 pigpio.h            /usr/local/include
install -m 0644 pigpiod_if.h        /usr/local/include
install -m 0644 pigpiod_if2.h       /usr/local/include
install -m 0755 -d                  /usr/local/lib
install -m 0755 libpigpio.so        /usr/local/lib
```

```
 GNU nano 2.2.6                File: gpio01.c

#include <stdio.h>
#include <pigpio.h>
/* ----------------------------------------------------------------------
   gcc -o gpio01 gpio01.c -lpigpio -lpthread -lrt
*/
#define GPIO_PIN 18

int main (int argc, char *argv[])
{
   int i=0;
   if (gpioInitialise()<0) return 1;
   gpioSetMode(GPIO_PIN, PI_OUTPUT);
   for (i=0;i<10;i++)
   {
      gpioWrite(GPIO_PIN, PI_OFF);
      usleep(500 * 1000);
      gpioWrite(GPIO_PIN, PI_ON);
      usleep(500 * 1000);
   }

   gpioTerminate();
}
```

```
root@raspberrypi:/home/pi/session06/gpio# gcc -o gpio01 gpio01.c -lpigpio -lpthread -lrt
root@raspberrypi:/home/pi/session06/gpio#
```

```
  GNU nano 2.2.6                                    File: freq_count_1.c

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <unistd.h>

#include <pigpio.h>

#define OPT_R_MIN 1
#define OPT_R_MAX 10
#define OPT_R_DEF 5

#define OPT_S_MIN 1
#define OPT_S_MAX 10
#define OPT_S_DEF 5

static volatile int g_pulse_count;
static volatile int g_reset_count;

static int g_gpio;
static uint32_t g_mask;

static int g_opt_r = OPT_R_DEF;
static int g_opt_s = OPT_S_DEF;
static int g_opt_t = 0;
```

```
void usage()
{
   fprintf
   (stderr,
      "\n" \
      "Usage: sudo ./freq_count_1 gpio ... [OPTION] ...\n" \
      "   -r value, sets refresh period in deciseconds, %d-%d, default %d\n" \
      "   -s value, sets sampling rate in micros, %d-%d, default %d\n" \
      "\nEXAMPLE\n" \
      "sudo ./freq_count_1 4 -r2 -s2\n" \
      "Monitor gpio 4.  Refresh every 0.2 seconds.  Sample rate 2 micros.\n" \
      "\n",
      OPT_R_MIN, OPT_R_MAX, OPT_R_DEF,
      OPT_S_MIN, OPT_S_MAX, OPT_S_DEF
   );
}
```

```c
void fatal(int show_usage, char *fmt, ...)
{
   char buf[128];
   va_list ap;

   va_start(ap, fmt);
   vsnprintf(buf, sizeof(buf), fmt, ap);
   va_end(ap);

   fprintf(stderr, "%s\n", buf);

   if (show_usage) usage();

   fflush(stderr);

   exit(EXIT_FAILURE);
}
```

```c
static int initOpts(int argc, char *argv[])
{
   int i, opt;

   while ((opt = getopt(argc, argv, "r:s:")) != -1)
   {
      i = -1;

      switch (opt)
      {
         case 'r':
            i = atoi(optarg);
            if ((i >= OPT_R_MIN) && (i <= OPT_R_MAX))
               g_opt_r = i;
            else fatal(1, "invalid -r option (%d)", i);
            break;

         case 's':
            i = atoi(optarg);
            if ((i >= OPT_S_MIN) && (i <= OPT_S_MAX))
               g_opt_s = i;
            else fatal(1, "invalid -s option (%d)", i);
            break;

        default: /* '?' */
           usage();
           exit(-1);
        }
    }
   return optind;
}
```

```
void edges(int gpio, int level, uint32_t tick)
{
   int g;

   if (g_reset_count)
   {
      g_reset_count = 0;
      g_pulse_count = 0;
   }

   /* only record low to high edges */
   if (level == 1) g_pulse_count++;
}
```

```
int main(int argc, char *argv[])
{
   int rest, g, mode;
   int count;

   /* command line parameters */

   rest = initOpts(argc, argv);

   if (rest < argc){
      g = atoi(argv[rest]);
      if ((g>=0) && (g<32))
      {
         g_gpio = g;
         g_mask |= (1<<g);
      }else fatal(1, "%d is not a valid g_gpio number\n", g);
   }else{
      fatal(1, "At least one gpio must be specified");
   }

   printf("Monitoring gpio");
   printf(" %d", g_gpio);
   printf("\nSample rate %d micros, refresh rate %d deciseconds\n",
      g_opt_s, g_opt_r);
```

```
   gpioCfgClock(g_opt_s, 1, 1);

   if (gpioInitialise()<0) return 1;

   /* monitor g_gpio level changes */

   gpioSetAlertFunc(g_gpio, edges);

   gpioSetMode(g_gpio, PI_INPUT);

   while (1)
   {
      count = g_pulse_count;

      g_reset_count = 1;

      printf(" %d=%d\n", g_gpio, count);

      gpioDelay(g_opt_r * 100000);
   }

   gpioTerminate();
}
```

خواندن مقدار یک سنسور آنالوگ توسط یک پایه ی دیجیتال:

Reading Analogue Sensors With One GPIO Pin

Unlike some other devices the Raspberry Pi does not have any analogue inputs. All 17 of its GPIO pins are digital. They can output high and low levels or read high and low levels. This is great for sensors that provide a digital input to the Pi but not so great if you want to use a sensor that doesn't.

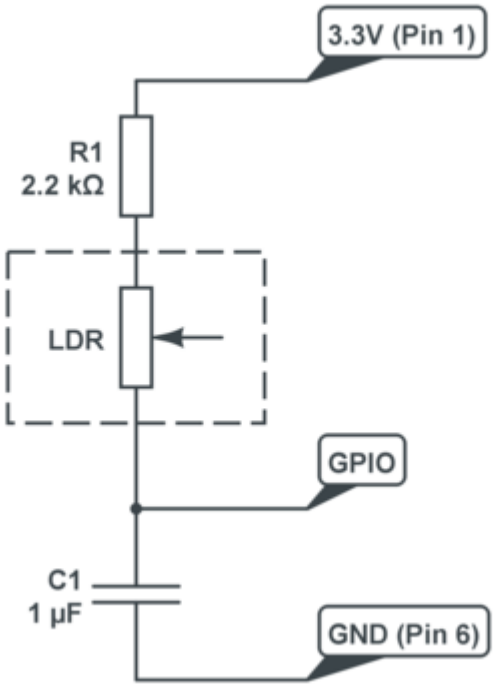
It uses a basic "RC" charging circuit (Wikipedia Article) which is often used as an introduction to electronics. In this circuit you place a Resistor in series with a Capacitor. When a voltage is applied across these components the voltage across the capacitor rises. The time it takes for the voltage to reach 63% of the maximum is equal to the resistance multiplied by the capacitance. When using a Light Dependent resistor this time will be proportional to the light level. This time is called the time constant :

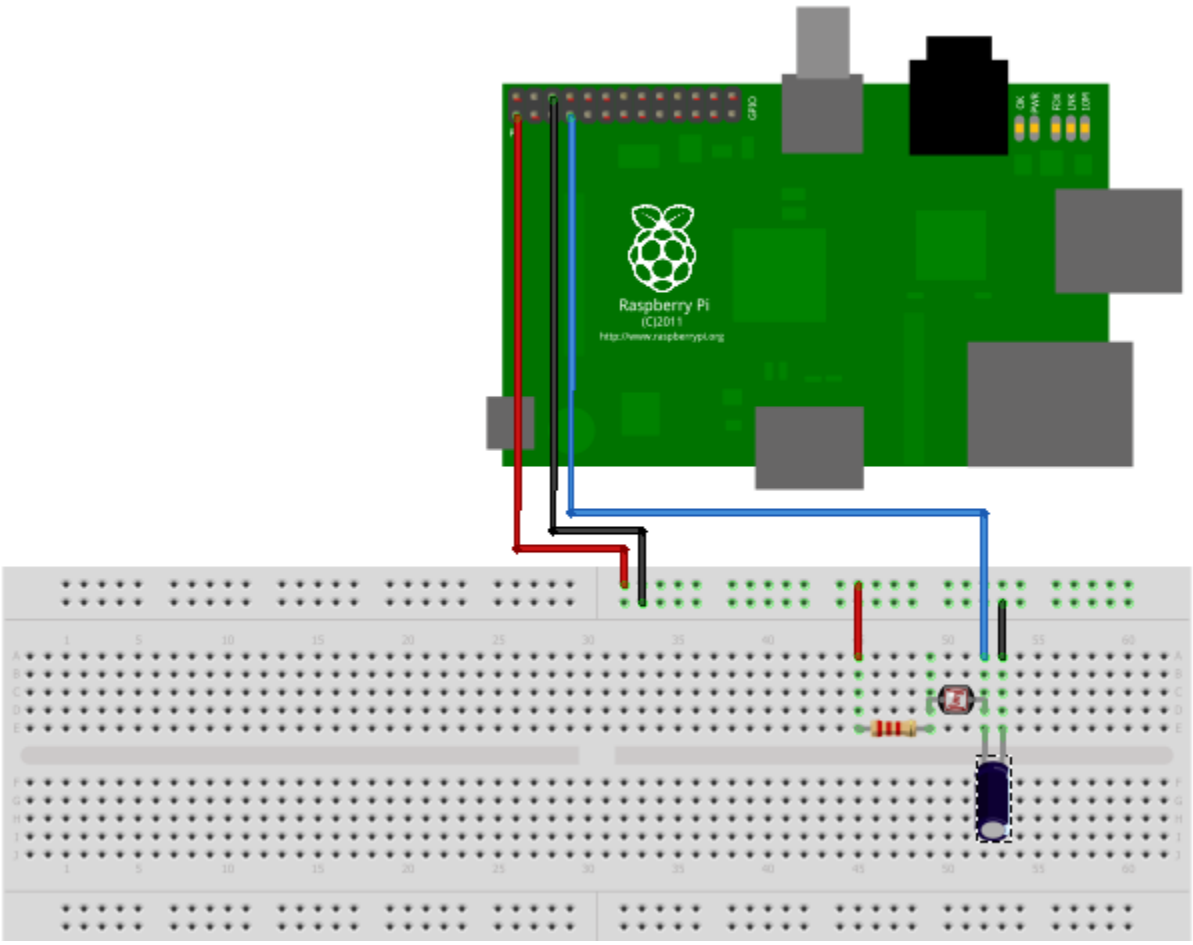
> t = RC
>
> where t is time,
> R is resistance (ohms)
> and C is capacitance (farads)

So the trick is to time how long it takes a point in the circuit the reach a voltage that is great enough to register as a "High" on a GPIO pin. This voltage is approximatey 2 volts, which is close enough to 63% of 3.3V for my liking. So the time it takes the circuit to change a GPIO input from Low to High is equal to 't'.

With a 10Kohm resistor and a 1uF capacitor t is equal to 10 milliseconds. In the dark our LDR may have a resistance of 1Mohm which would give a time of 1 second. You can calculate other values using an online time constant calculator.

In order to guarantee there is always some resistance between 3.3V and the GPIO pin I inserted a 2.2Kohm resistor in series with the LDR.


3.3V (Pin 1)
R1
2.2 kΩ
LDR
GPIO
C1
1 µF
GND (Pin 6)


Raspberry Pi
(C)2011

Theory

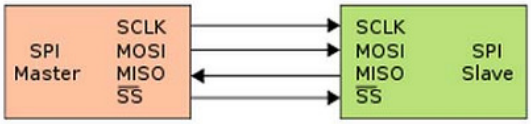
Here is the sequence of events :

- Set the GPIO pin as an output and set it Low. This discharges any charge in the capacitor and ensures that both sides of the capacitor are 0V.
- Set the GPIO pin as an input. This starts a flow of current through the resistors and through the capacitor to ground. The voltage across the capacitor starts to rise. The time it takes is proportional to the resistance of the LDR.
- Monitor the GPIO pin and read its value. Increment a counter while we wait.
- At some point the capacitor voltage will increase enough to be considered as a High by the GPIO pin (approx 2v). The time taken is proportional to the light level seen by the LDR.
- Set the GPIO pin as an output and repeat the process as required.

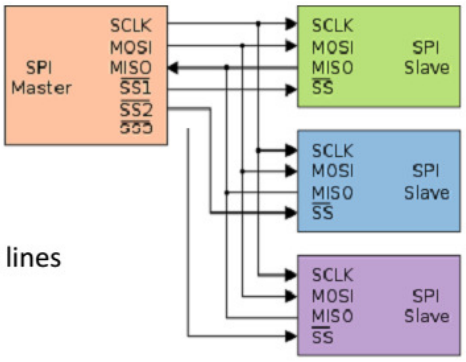
آشنایی با SPI

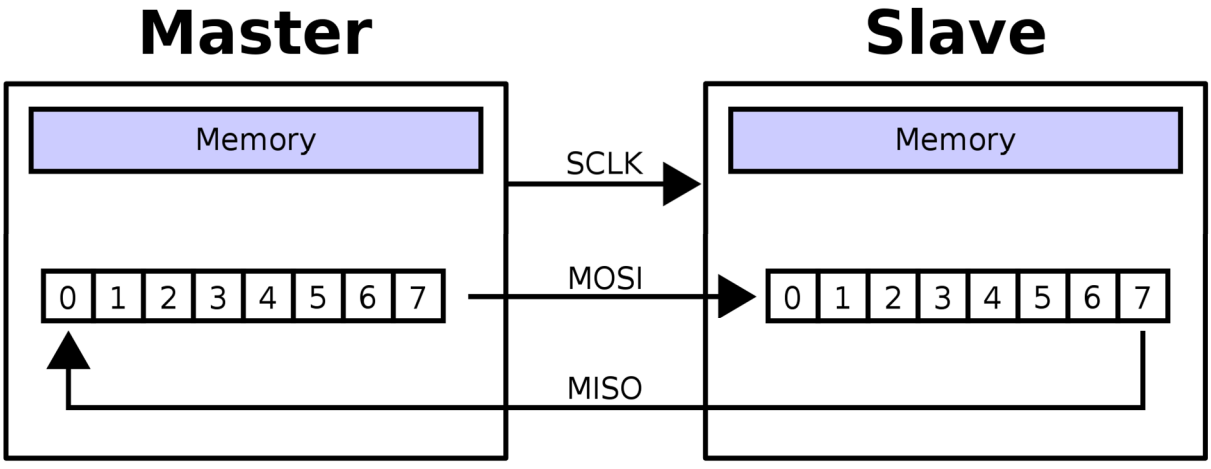
SPI

(Serial Peripheral Interface - Motorola)



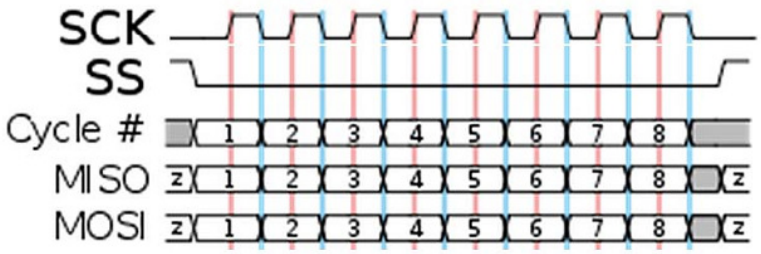
- Two types of devices, masters and slaves.
- We'll consider only one master, but multiple slaves.
- Signals
  - SCLK: Serial CLocK, set by Master
  - MOSI: Master Out, Slave In
  - MISO: Master In, Slave Out
  - ~SS: Slave Select
    - Each slave gets its own slave select (other lines are shared)
    - Pulling line low selects slave





SPI and the clock
(intro)

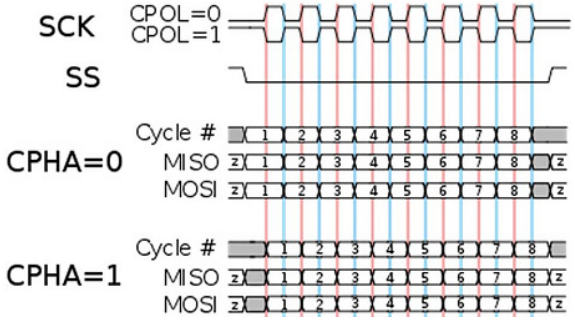
- Pull slave select line low to select device.
- First bit of data gets put on MISO and MOSI (so a byte goes both ways)
- Data gets shifted out (typically 8 bits, but not necessarily)
  - The data gets put on bus on falling edge of clock.
  - The data gets read on the rising edge of clock.



SPI and the clock
(the hard truth)

Unfortunately, clock can be set many ways as determined by clock polarity and phase.

- CPOL=0: Base value of the clock is 0
  - CPHA=0: Data read on rising edge, put on bus on falling edge of SCLK. (i.e., clock is low). (Case from previous slide)
  - CPHA=1: Data read on falling edge, put on bus on rising edge (i.e., clock is high).
- CPOL=1: Base value of the clock is 1
  - CPHA=0: Data read on falling edge, put on bus on rising edge (i.e., clock is high).
  - CPHA=1: Data read on rising edge, put on bus on falling edge (i.e., clock is low).



```
root@raspberrypi:/home/pi# nano /boot/config.txt
```

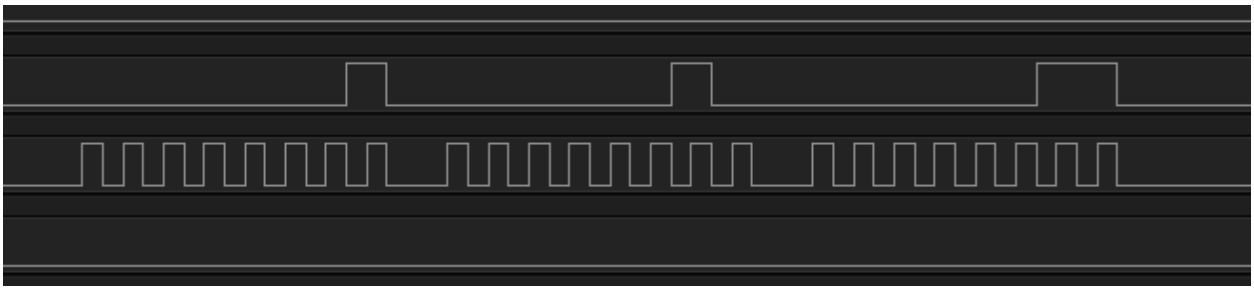
```
  GNU nano 2.2.6                File: /boot/config.txt


#uncomment to overclock the arm. 700 MHz is the default.
#arm_freq=800

# Uncomment some or all of these to enable the optional hardware interfaces
#dtparam=i2c_arm=on
#dtparam=i2s=on
dtparam=spi=on

# Uncomment this to enable the lirc-rpi module
#dtoverlay=lirc-rpi

# Additional overlays and parameters are documented /boot/overlays/README
```

```
root@raspberrypi:/home/pi# reboot
```

```
root@raspberrypi:/home/pi# ls /dev/ | grep spi
spidev0.0
spidev0.1
root@raspberrypi:/home/pi#
```

تست:

```
root@raspberrypi:/home/pi/session06/spi# echo -ne "\x01\x02\x03" > /dev/spidev0.0
root@raspberrypi:/home/pi/session06/spi#
```



```
  GNU nano 2.2.6                                          File: spitest.c

#include <stdint.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/types.h>
#include <linux/spi/spidev.h>

#define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))

static void pabort(const char *s)
{
        perror(s);
        abort();
}
static const char *device = "/dev/spidev0.0";
static uint8_t mode;
static uint8_t bits = 8;
static uint32_t speed = 500000;
static uint16_t delay;
```

```c
static void transfer(int fd)
{
        int ret;
        uint8_t tx[] = {
                0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
                0x40, 0x00, 0x00, 0x00, 0x00, 0x95,
                0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
                0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
                0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
                0xDE, 0xAD, 0xBE, 0xEF, 0xBA, 0xAD,
                0xF0, 0x0D,
        };
        uint8_t rx[ARRAY_SIZE(tx)] = {0, };
        struct spi_ioc_transfer tr = {
                .tx_buf = (unsigned long)tx,
                .rx_buf = (unsigned long)rx,
                .len = ARRAY_SIZE(tx),
                .delay_usecs = delay,
                .speed_hz = speed,
                .bits_per_word = bits,
        };

        ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
        if (ret < 1)
                pabort("can't send spi message");

        for (ret = 0; ret < ARRAY_SIZE(tx); ret++) {
                if (!(ret % 6))
                        puts("");
                printf("%.2X ", rx[ret]);
        }
        puts("");
}
```

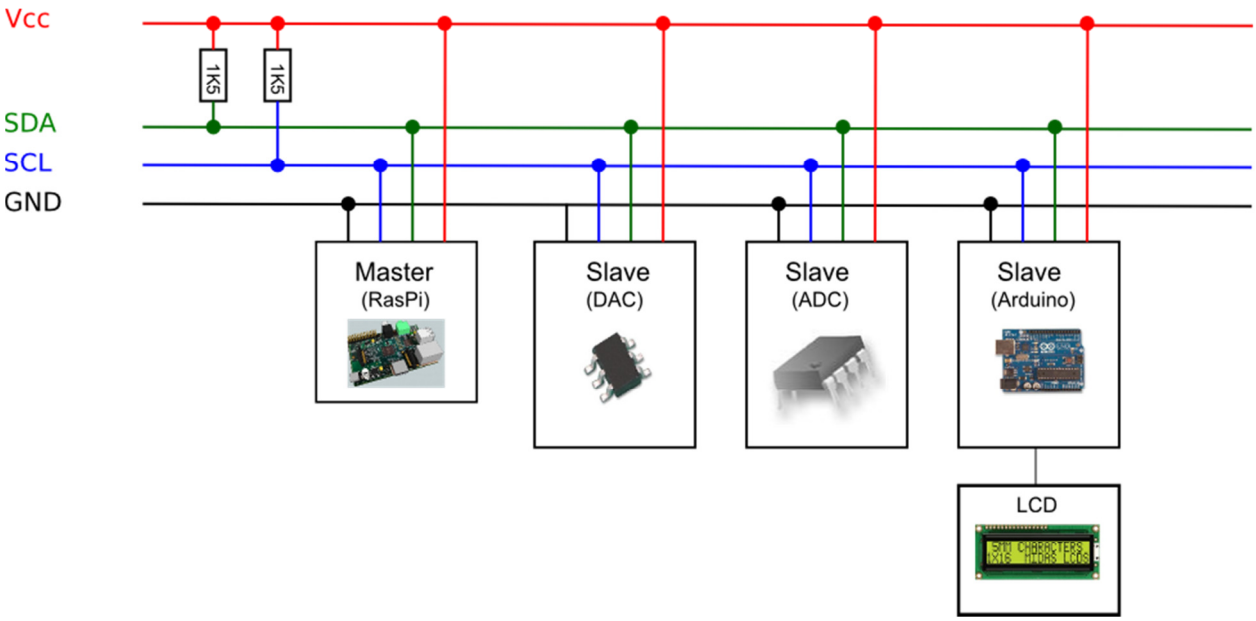
```c
static void print_usage(const char *prog)
{
        printf("Usage: %s [-DsbdlHOLC3]\n", prog);
        puts("  -D --device   device to use (default /dev/spidev1.1)\n"
             "  -s --speed    max speed (Hz)\n"
             "  -d --delay    delay (usec)\n"
             "  -b --bpw      bits per word \n"
             "  -l --loop     loopback\n"
             "  -H --cpha     clock phase\n"
             "  -O --cpol     clock polarity\n"
             "  -L --lsb      least significant bit first\n"
             "  -C --cs-high  chip select active high\n"
             "  -3 --3wire    SI/SO signals shared\n");
        exit(1);
}
```

```c
static void parse_opts(int argc, char *argv[])
{
        while (1) {
                static const struct option lopts[] = {
                        { "device",  1, 0, 'D' },
                        { "speed",   1, 0, 's' },
                        { "delay",   1, 0, 'd' },
                        { "bpw",     1, 0, 'b' },
                        { "loop",    0, 0, 'l' },
                        { "cpha",    0, 0, 'H' },
                        { "cpol",    0, 0, 'O' },
                        { "lsb",     0, 0, 'L' },
                        { "cs-high", 0, 0, 'C' },
                        { "3wire",   0, 0, '3' },
                        { "no-cs",   0, 0, 'N' },
                        { "ready",   0, 0, 'R' },
                        { NULL, 0, 0, 0 },
                };
                int c;

                c = getopt_long(argc, argv, "D:s:d:b:lHOLC3NR", lopts, NULL);
```

```
		if (c == -1)
			break;
		switch (c) {
		case 'D':
			device = optarg;
			break;
		case 's':
			speed = atoi(optarg);
			break;
		case 'd':
			delay = atoi(optarg);
			break;
		case 'b':
			bits = atoi(optarg);
			break;
		case 'l':
			mode |= SPI_LOOP;
			break;
		case 'H':
			mode |= SPI_CPHA;
			break;
		case 'O':
			mode |= SPI_CPOL;
			break;
		case 'L':
			mode |= SPI_LSB_FIRST;
			break;
		case 'C':
			mode |= SPI_CS_HIGH;
			break;
		case '3':
			mode |= SPI_3WIRE;
			break;
		case 'N':
			mode |= SPI_NO_CS;
			break;
		case 'R':
			mode |= SPI_READY;
			break;
		default:
			print_usage(argv[0]);
			break;
		}
	}
```

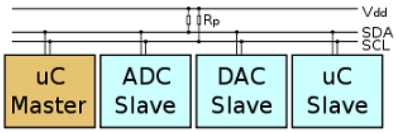
```c
int main(int argc, char *argv[])
{
        int ret = 0;
        int fd;
        parse_opts(argc, argv);

        fd = open(device, O_RDWR);
        if (fd < 0)   pabort("can't open device");
        /* spi mode */
        ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
        if (ret == -1)   pabort("can't set spi mode");

        ret = ioctl(fd, SPI_IOC_RD_MODE, &mode);
        if (ret == -1)   pabort("can't get spi mode");
        /* bits per word */
        ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
        if (ret == -1)   pabort("can't set bits per word");

        ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
        if (ret == -1)   pabort("can't get bits per word");

        /* max speed hz */
        ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
        if (ret == -1)   pabort("can't set max speed hz");

        ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
        if (ret == -1)   pabort("can't get max speed hz");

        printf("spi mode: %d\n", mode);
        printf("bits per word: %d\n", bits);
        printf("max speed: %d Hz (%d KHz)\n", speed, speed/1000);

        transfer(fd);
        close(fd);
        return ret;
}
```

آشنایی با i2c

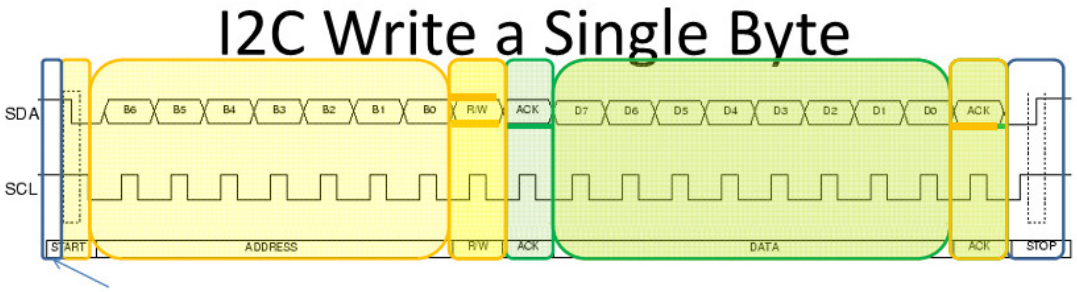


I2C or I²C

(Inter-Integrated Circuit – Philips)



- As with SPI a master-slave system.
- Also called a 2-wire bus.
  It Has only clock and data, with pull-up resistors (Rp in diagram).
- Lines can be pulled low by any device, and are high when all devices release them.
- There are no “slave-select” lines – instead the devices have “addresses” that are sent as part of the transmission protocol.
- Four max speeds (100 kbS (*standard*), 400 kbS (*fast*), 1 MbS (*fast plus*), and 3.4 MbS (*high-speed*)

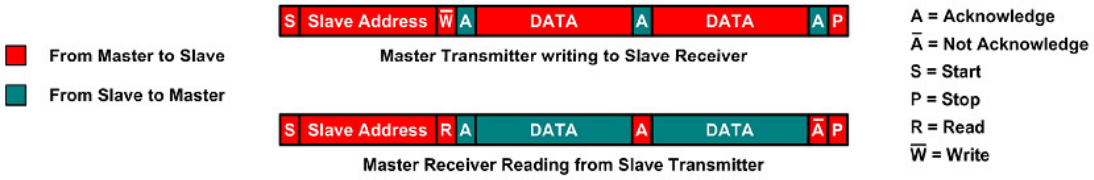


1. All: allow SDA, SCL start high
2. Master: SDA low to signal start
3. Master: Send out SCL, and 7 bit address followed by 0 (~W) on SDA
4. Slave: Pull SDA low to signify ACKnowledge
5. Master: Send out 8 data bits on SDA
6. Slave: Ack
7. All: allow SDA to go high when SCL is high (stop)

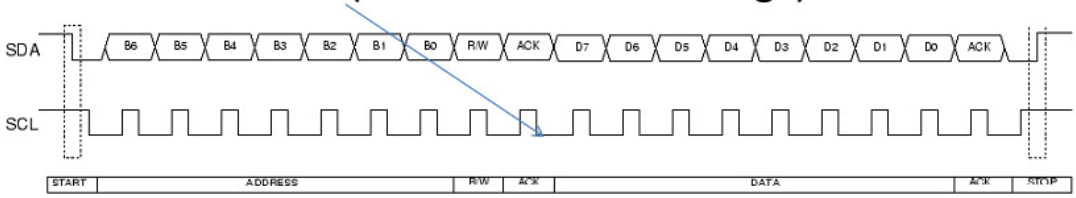
- **For "Read",**
  3. Master: Address following by 1 (R) on SDA
  5. Slave: Send out 8 data bits on SDA
  6. Master: Ack

Other Features

- You can transfer multiple bytes in a row



- At end of transfer, slave can hold SCL low to slow transfer down (called “clock-stretching”)



- Any device that malfunctions can disable bus.

```
root@raspberrypi:/home/pi# nano /boot/config.txt
root@raspberrypi:/home/pi#
```

```
  GNU nano 2.2.6                File: /boot/config.txt


#uncomment to overclock the arm. 700 MHz is the default.
#arm_freq=800

# Uncomment some or all of these to enable the optional hardware interfaces
dtparam=i2c_arm=on
dtparam=i2s=on
dtparam=spi=on

# Uncomment this to enable the lirc-rpi module
#dtoverlay=lirc-rpi

# Additional overlays and parameters are documented /boot/overlays/README
```

```
root@raspberrypi:/home/pi# reboot
```

```
root@raspberrypi:/home/pi# ls /dev/ | grep i2c
root@raspberrypi:/home/pi# modprobe i2c-dev
root@raspberrypi:/home/pi# ls /dev/ | grep i2c
i2c-1
root@raspberrypi:/home/pi#
```